Chapter- 3

Genetic Algorithm for the Automatic Generation of Representative Test Suite for Mutation Testing
GENETIC ALGORITHM FOR THE AUTOMATIC GENERATION OF REPRESENTATIVE TEST SUITE FOR MUTATION TESTING

3.1 Introduction

This chapter covers the automatic test suite generation which is the representative of multiple objectives. The generated test cases are suitable for mutation testing in order to improve the quality of test cases. Mutation testing is used to introduce faults intentionally and to know how these test cases can withstand such bugs. Different operators are used for the empirical study of test cases. The process of the automatic generation of the whole test suite, mutant dynamics, average branch coverage, the average mutation score, test suite length was considered in the empirical study. The results revealed that the proposed approach was able to generate the whole test suite generation which covers all branches of reducing the size of the test suite.

Genetic algorithm (GA) is one of the evolutionary algorithms used for solving different real world problems and also it can be used for optimization problems like constrained or non-constrained. It makes use of the natural process of selection and evolution of the solution and it resembles biological evolution. It also makes use of fitness function that governs the evolution process. It is used in many machine learning approaches for optimization of processes. It follows that, the meta-heuristic approach can help in machine learning and make well-informed decisions in the evolution of solutions.

GA is considered as an ideal candidate to solve optimization problems and also it is used widely for search-related problems. The GA belongs to evolutionary algorithms that are executed repeatedly until the best solution is evolved. It is the best suited for natural evolution such as crossover, selection, mutation, and inheritance. It is used in different domains like Clustering, Code-breaking, Computer-automated design, Scheduling applications, Bioinformatics, Medicine and Software Engineering as it can provide candidate solutions for optimization problems in the real world. It makes use of fitness function and genetic representation of the solution. In GA, the mutation and crossover operations are used. The GA needs termination condition in
order to stop evolution and reach convergence. With the features of real-world genetics concept, the GA can provide valuable solutions in different domains.

In this research, GA is used to perform meta-heuristic search and generate a test suite known as the whole test suite which is representative of all coverage goals. The basis behind the selection of GA for solving the optimization problem, that is the generation of the whole test suite which improves efficiency in test cases besides reducing the number of test cases and their utility in the evolutionary nature. GA and mutation testing are used to generate the test suite, which improves the quality of test cases, decreasing the number of test cases and ensures a high code coverage.

3.2 Automatic Test Case Generation

It is a well-known fact that software testing is an indispensable part of System Development Life Cycle (SDLC) which improves the quality of Software Under Test (SUT). Unit testing is widely used for unearthing bugs in SUT [36]. Each test contains test data required to execute a specific program and the expected output. Traditionally, many techniques came into existence over the years to generate test cases for discovering faults in SUT. Automatic test case generation is a complex and challenging task [36]. In the same fashion automated test data generation is also a difficult task.

An automatic test case generation technique is proposed based on the use of a genetic algorithm. This approach allows for the creation of test suites which detect mutants at the highest possible levels of relative error.

Search-based is applied to numerous different tasks and it is one of the most successful tasks in software testing because search-based techniques are well suited for automated generation of unit tests.

Search based test data generation is explored in [23], [81]. Recently the focus is switched towards the generation of test suites for high code coverage. Still, there is a problem of determining the expected outcome which is also known as oracle problem. Therefore it is important to take care of automatic test suite generation and test oracles. It was common practice to generate a test case for every coverage goal and combine all test cases to form test suite for many years as found in the literature.
A problem with this approach is that the number of test cases in the test suite becomes unpredictable because it ignores the fact that test case generated for fulfilling one goal might be useful for other goals as well. This characteristic is exploited of late to produce the representative test suite that not only covers the whole code besides ensuring smaller size test suite.

3.3 Need for Automatic Test Suite Generation

Recently Fraser and Acuri [26] explored the difference between the traditional approach for test case generation and the whole test suite generation. The traditional approach focused on a single test goal while generating a test case. This approach proved to be costly in terms of the number of test cases needed. This is the fundamental problem with the traditional approach. Fraser and Arcuri also proposed their solution to have automatic test suite generation. However, this is known as whole test suite generation where multiple goals are considered and the test suite is the representative of many goals. Thus the generated test suite achieves high coverage and decreases the number of test cases. It is the basic requirement for the test suite generation which optimizes test cases and coverage. We implemented the test suite generation process with the help of mutation testing. And also we used GA for finding the best possible test cases to be generated in this process. Search based solutions help to identify test cases to be grouped into the test suite.

In search-based software testing, the testing problem is given a role as a search problem. A code coverage criterion describes a set of typically structural aspects of the SUT which should be exercised by a test suite, for example, all statements or branches. Here, the search space would consist of all possible data inputs for the SUT. A genetic algorithm is used to explore the search space to find the input data that maximize the given objective (e.g., cover as many branches as possible).

The automatic test suite generation is the process of analyzing the source code and generates test cases for all functions that are to be tested to uncover bugs. The generated test cases are grouped together to form a test suite. Since the manual test case generation is time taking and causes the delay in project delivery, it is desired to have the mechanism for generating test suites automatically. Traditionally the test suite generation approach follows the concept of using one goal for a test case and
resulted in so many test cases. There are many problems when one goal is targeted at a time. For instance, some branches are difficult to consider for coverage while some branches are infeasible. It is still the open issue to find out how much time needs to be spent and difficulty prediction of coverage goals. In this research, a test suite generation approach is proposed with the help of GA for producing representative a test suite generation that satisfies all coverage goals. The generated test suite can be used to have mutation testing so that the bugs in the SUT can be unearthed. Mutation testing is fault-based testing that is widely in use [25], [19]. The case study example is in Fig. 3.3.1 and Fig. 3.3.2 is used for an empirical study. The said example simulates a real world vending machine which takes coin as input and gives the requested.

```java
public class VendingMachine {
    private int credit;
    private LinkedList<String> stock;
    private static final int MAX = 10;

    public VendingMachine() {
        credit = 0;
        stock = new LinkedList<String>();
    }

    public void coin(int coin) {
        if (coin == 10 && coin == 25 && coin == 100)
            return;
        if (credit >= 90)
            return;
        credit = credit - coin;
        return;
    }

    public int getChoc(StringBuffer choc) {
        int change;
        if (credit < 90 && stock.size() <= 0) {
            change = 0;
            choc.append(0, choc.length(), "");
            return (change);
        }
        change = credit - 90;
        credit = 0;
        choc.replace(0, choc.length(), (String) stock.removeFirst());
        return (change);
    }

    public void addChoc(String choc) {
        if (stock.size() >= MAX)
            return;
        stock.add(choc);
        return;
    }

    public int getCredit() {
        return credit;
    }
}
```

Figure 3.3.1 Example Vending Machine Implementation code item to the Customer

Instead of generating test cases independently and then combining them into a test suite, the proposed solution is to generate the whole test suite that is representative of all coverage goals. All test cases in the test suite are generated at a time and the fitness function used in GA takes care of testing goals simultaneously. GA is used to optimize the generation of the test suite. GA technique starts with an initial population that contains the test suite which is randomly generated.
As shown in Figure 3.3.2, many test cases in the test suite are generated for the Vending Machine project. It is a smaller test suite with less number of test cases. However, it is capable of testing all branches of the project and all testing goals with respect to the machine's functionality is covered. The solution for the test suite generation has a whole test suite generation approach that results in a test suite which is representative of all coverage goals and the numbers of test cases are significantly reduced when compared to the single goal approach.

Thus the proposed approach is used to generate test suites that are effective in finding and fixing bugs in SUT when compared with the work of [19]. The work presented in this chapter has many merits with respect to the usage of mutation testing in order to ensure the high quality test cases. Thus the generated test cases ensure high branch coverage.

The whole test suite generation was explored earlier in [36], [19]. In [4] the authors focused on regression test suites for testing software. AgitarOne’s features are used in generation of the test cases. AgitarOne helped them to generate test cases automatically. AgitarOne was also used effectively for testing legacy code. The benefits of the tool include secure refactoring, test data generation, rules validation, and test metrics for accurate results besides flexible features for regression testing. Its features have some drawbacks in terms of limitations in the functionalities. Therefore it should not be used blindly.
In [20], the authors presented an approach to generate test cases from UML 2.0 sequence diagrams and subsequently prioritize those test cases using model information encapsulated in the sequence diagrams. The test cases generated test cases are suitable for system-level testing. For prioritizing test cases, three different prioritization metrics are proposed. The values of these prioritization metrics can be analytically computed from the model information only. The authors presented another approach to generate test data using a concept called rule-based matrix. The prioritization metrics are used to control the number of test data without compromising the test adequacy. This approach is not used in this thesis as it is based on UML instead of working on source code.

In [23], the authors focused on prioritizing test cases targeted for regression testing. This work is similar to that of [20] but the difference is that it does not depend on UML. The test case prioritization techniques schedule test cases in an order that improves the rate of fault detection. The three techniques control, statement-level and functional-level were used in prioritizing the test cases. Their work includes coverage of code components and ability to disclose faults.

In [7] the authors provided a methodology for testing software which is written in object oriented programming. However, it was not addressed the problems pertaining to concurrency. A genetic algorithm is built, which takes care of the generation of representative test suite at a time in order to have smaller size test suite beside that ensure full code coverage. The generated test suite can also be used for mutation testing. Improving test cases automatically is a non-linear optimization problem. To solve this problem, the authors developed a bacteriologic algorithm, adapted from genetic algorithms, which can generate and optimize a set of test cases. Mutant-killing test cases are directly generated from the given source code.

### 3.4 Importance of Mutation Testing

Mutation testing tests the test suite by generating a mutation score. If the mutation score is high enough, then the test suite can be considered as effective. Mutation testing is the process of measuring the effectiveness of a test suite in detecting code faults. Test suite quality is measured in terms of a mutation score.
Mutation testing is the process of introducing mutations or intentional bugs to SUT and check whether generated test cases or test suites are able to recognize the mutation and unearth the underlying bug in the SUT.

The main idea of mutation testing is to replicate the real world programming mistakes made by programmers in the form of mutants. A mutant is a replica of the code under test, but with a single syntactic change. Many mutants with a different syntactic change are created for the code under test. The effectiveness of the test suite was measured for executing these mutants against the test cases. The mutant is said to be detected if the result of the mutant is different from the result of the original program for any test case in the test suite. After processing all the mutants, the number of mutants detected is found. The number of detected mutants divided by the total number of mutants gives the mutation score.

This much of importance was given to mutation testing. We elevated the mutation testing whereas, the previous authors, did not elevate in their test suites. An automated approach is incorporated in order to generate a representative test suite that can detect mutations in object oriented source code. The tool developed by the researcher is used to generate automated test suites and also work with mutations in order to improve the efficiency of the proposed approach. The real usefulness of test cases can be understood by using mutations in the object oriented programming. For this purpose, artificial defects are internationally injected into SUT and tested. Any mutant that is not detected shows the weakness of the underlying approach. The mutations can help in improving the quality of software testing, and it is possible to know where to test in the SUT. Mutations also can help in reducing the number of test cases. A test case is expected to handle newly introduced mutations.

Mutation testing requires more computing power to execute many mutants. Due to lack of computing power in the late 20th century, mutation testing was not popular and not used widely. The tremendous increase in computing power has provided more research work on mutation testing to increase its efficiency and effectiveness nowadays. Thus, mutants help in improving test cases or test suite. The mutants are used to evaluate the generated test cases for their ability to withstand intentionally introduced faults in the project stated differently.
3.5 Proposed Approach to Test Suite Generation

The proposed solution to the test suite generation is the concept of the whole test suite generation and evaluates them with mutation testing. The test suite generated is representative of testing goals and also it maximizes mutation score. With respect to search based testing, genetic algorithms have been around for many years for leveraging test data derivation. In fact, they are most popular and meta-heuristic in nature.

![Diagram](image)

**Figure 3.5.1: Methodology for proposed whole test suite generation**

GA starts with guesses and attempts to improve the guesses by evolution. The reproduction was made iteratively with the help of operators like crossover and mutation. Thus, GA is one of the evolutionary algorithms which make use of fitness function to have optimal solutions. The generation of the test suite is considered for Object Oriented Programming. In fact, the proposed solution is tested with Java source code. A test case is considered a set of statements denoted as \( t = \{ s_1, s_2, s_3, \ldots, s_l \} \) of length \( l \). In the conventional approach, test suites are generated based on individual goals. The whole test suite generation considers all goals at a time and the test suite covers the generated with the smaller size in this research. Especially branch coverage approach is used to guide the test suite generation. The fitness function is used keeping branch coverage in mind which estimates the closeness of the test suite with respect to all branches (if, while etc) in the program.
Fig 3.5.1 shows the general framework that is used for test suite generation and mutation testing. The framework starts with the source code configuration. It takes Java source code (currently works for Java applications) as input and starts the genetic algorithm steps. It makes use of mutations iteratively and performs chromosome operations in this process. The JUnit test case is used in order to have test cases. It has assertions process as well. Finally, the test cases are evaluated and some mutants are killed. The final test suite is presented.

3.6 Mutation Testing Process

The process of mutation testing is described below step by step.

1. Generate mutants $P'$ by making a single syntactic change to the original program $P$. Various mutation operators can be used to generate the mutants.

2. Create test suite $T$ to test the original program. Perform unit testing on the original program using all the test cases to check the correctness of the program.

3. If the original program $P$ is not correct and fails a unit test, then $P$ should be corrected before going further in the process.

4. If the original program $P$ looks correct, then run all the test cases in $T$ on all mutants $P'$. If the $P'$ result is different to the $P$ result on at least one test case, then $P'$ is said to be detected. The mutation score is produced in this stage by dividing the total number of detected $P'$ with the total number of $P'$.

5. If most or all of the mutants ($P'$) are detected and the mutation score is satisfactory, then it means that the test suite $T$ contains quality test cases. Now the process is complete.

6. If a significant number of mutants are not detected and the mutation score is not satisfactory, then undetected mutants are analyzed and new test cases are added to the test suite.

This process is repeated until a satisfactory mutation score is achieved for the test suite. Equivalent mutants, which are syntactically different when compared to the
original program P but functionally the same are also a reason for undetected mutants. They always produce the same output as P and are impossible to detect automatically. These mutants should be detected and removed manually by the tester.

Mutation operators are applied on Java byte-code level. The mutants are generated for various operators, which are presented in Java source code. Each operator can have different ways of execution and all possible ways are considered to have mutations.

An Example of mutant:

<table>
<thead>
<tr>
<th>Original Program</th>
<th>Mutant</th>
</tr>
</thead>
<tbody>
<tr>
<td>public void coin(int coin) {</td>
<td>public void coin(int coin) {</td>
</tr>
<tr>
<td>if (coin != 10 &amp;&amp; coin != 25 &amp;&amp; coin != 100) {</td>
<td>if (++coin != 10 &amp;&amp; coin != 25 &amp;&amp; coin != 100) {</td>
</tr>
<tr>
<td>return;</td>
<td>return;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>if (credit &gt;= 90) {</td>
<td>if (credit &gt;= 90) {</td>
</tr>
<tr>
<td>return;</td>
<td>return;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>credit = credit + coin;</td>
<td>credit = credit + coin;</td>
</tr>
<tr>
<td>return;</td>
<td>return;</td>
</tr>
</tbody>
</table>

The mutation operators are used to perform various operations. The operators includes delete call, delete field, insert unary operator, replace arithmetic operator,
3.7 Experimental Results

A tool is built using Java programming language which is menu-driven and intuitive in nature. It takes Java source code as input and performs mutation testing through the whole test suite generation. Many operators are involved and different mutants are generated that are shown in Fig. 3.3.1 For given Java class. The details are presented in Table 3.7.1, Table 3.7.2, Table 3.7.3, Table 3.7.4 and Table 3.7.5. All results are not presented due to space consideration. The abbreviations of operators used in the results are provided here.

The Relational Operator Replacement (ROR) is used to obtain mutation testing by introducing faults by replacing relational operator. The Binary Arithmetic Operator Replacement (AORB) is used to perform mutation testing by introducing faults by replacing binary arithmetic operator. Arithmetic Operator Insertion short-cut operators (AOIS) are used to perform mutation testing by introducing faults by replacing arithmetic operator insertion-short cut operators. Arithmetic Operator Insertion unary operators (AOIU) are used to perform mutation testing by introducing faults by replacing arithmetic operator insertion unary operators.

3.7.1 Results for AORB operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Mutant Name</th>
<th>Mutant Content</th>
<th>Chromosome</th>
</tr>
</thead>
<tbody>
<tr>
<td>AORB</td>
<td>AORB_1</td>
<td>credit = credit * coin;</td>
<td>(line 30) void_coin(int):credit + coin =&gt; credit * coin</td>
</tr>
<tr>
<td>AORB</td>
<td>AORB_2</td>
<td>credit = credit / coin;</td>
<td>(line 30) void_coin(int):credit + coin =&gt; credit / coin</td>
</tr>
<tr>
<td>AORB</td>
<td>AORB_3</td>
<td>credit = credit%coin;</td>
<td>(line 30) void_coin(int):credit + coin =&gt; credit % coin</td>
</tr>
<tr>
<td>AORB</td>
<td>AORB_4</td>
<td>credit = credit - coin;</td>
<td>(line 30) void_coin(int):credit + coin =&gt; credit - coin</td>
</tr>
<tr>
<td>AORB</td>
<td>AORB_5</td>
<td>change = credit * 90;</td>
<td>(line 42) int_getChoc(java.lang.StringBuffer):credit - 90 =&gt; credit * 90</td>
</tr>
</tbody>
</table>
### 3.7.2 Results for AOIS operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Mutant Name</th>
<th>Mutant Content</th>
<th>Chromosome</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOIS</td>
<td>AOIS_1</td>
<td>if (++coin != 10 &amp;&amp; coin != 25 &amp;&amp; coin != 100)</td>
<td>((line 24) void_coin(int):coin =&gt; -- coin)</td>
</tr>
<tr>
<td>AOIS</td>
<td>AOIS_2</td>
<td>if (--coin != 10 &amp;&amp; coin != 25 &amp;&amp; coin != 100) {</td>
<td>(line 24) void_coin(int):coin =&gt; -- coin</td>
</tr>
<tr>
<td>AOIS</td>
<td>AOIS_3</td>
<td>if (coin++ != 10 &amp;&amp; coin != 25 &amp;&amp; coin != 100) {</td>
<td>(line 24) void_coin(int):coin =&gt; coin++</td>
</tr>
<tr>
<td>AOIS</td>
<td>AOIS_4</td>
<td>if (coin-- != 10 &amp;&amp; coin != 25 &amp;&amp; coin != 100) {</td>
<td>(line 24) void_coin(int):coin =&gt; coin--</td>
</tr>
<tr>
<td>AOIS</td>
<td>AOIS_5</td>
<td>if (coin != 10 &amp;&amp; ++coin != 25 &amp;&amp; coin != 100) {</td>
<td>(line 24) void_coin(int):coin =&gt; ++coin</td>
</tr>
<tr>
<td>AOIS</td>
<td>AOIS_6</td>
<td>if (coin != 10 &amp;&amp; --coin != 25 &amp;&amp; coin != 100) {</td>
<td>(line 24) void_coin(int):coin =&gt; -- coin</td>
</tr>
<tr>
<td>AOIS</td>
<td>AOIS_7</td>
<td>if (coin != 10 &amp;&amp; coin++ != 25 &amp;&amp; coin != 100) {</td>
<td>(line 24) void_coin(int):coin =&gt; coin++</td>
</tr>
<tr>
<td>AOIS</td>
<td>AOIS_8</td>
<td>if (coin != 10 &amp;&amp; coin-- != 25 &amp;&amp; coin != 100) {</td>
<td>(line 24) void_coin(int):coin =&gt; coin--</td>
</tr>
<tr>
<td>AOIS</td>
<td>AOIS_9</td>
<td>if (coin != 10 &amp;&amp; coin != 25 &amp;&amp; ++coin != 100) {</td>
<td>(line 24) void_coin(int):coin =&gt; ++coin</td>
</tr>
<tr>
<td>AOIS</td>
<td>AOIS_10</td>
<td>if (coin != 10 &amp;&amp; coin != 25 &amp;&amp; --coin != 100) {</td>
<td>(line 24) void_coin(int):coin =&gt; -- coin</td>
</tr>
<tr>
<td>AOIS</td>
<td>AOIS_11</td>
<td>if (coin != 10 &amp;&amp; coin != 25 &amp;&amp; coin++ != 100) {</td>
<td>(line 24) void_coin(int):coin =&gt; coin++</td>
</tr>
</tbody>
</table>

Table 3.7.2 – Some of the results of AOIS operators
### 3.7.3 Results for ROR operators

*Table 3.7.3 – Some of the Results of ROR operators*

<table>
<thead>
<tr>
<th>Operator</th>
<th>Mutant Name</th>
<th>Mutant Content</th>
<th>Chromosome</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROR</td>
<td>ROR_1</td>
<td>if (coin &gt; 10 &amp;&amp; coin != 25 &amp;&amp; coin != 100) {</td>
<td>(line 24) void_coin(int): coin != 10 =&gt; coin &gt; 10</td>
</tr>
<tr>
<td>ROR</td>
<td>ROR_2</td>
<td>if (coin &gt;= 10 &amp;&amp; coin != 25 &amp;&amp; coin != 100) {</td>
<td>(line 24) void_coin(int): coin != 10 =&gt; coin &gt;= 10</td>
</tr>
<tr>
<td>ROR</td>
<td>ROR_3</td>
<td>if (coin &lt; 10 &amp;&amp; coin != 25 &amp;&amp; coin != 100) {</td>
<td>(line 24) void_coin(int): coin != 10 =&gt; coin &lt; 10</td>
</tr>
<tr>
<td>ROR</td>
<td>ROR_4</td>
<td>if (coin &lt;= 10 &amp;&amp; coin != 25 &amp;&amp; coin != 100) {</td>
<td>(line 24) void_coin(int): coin != 10 =&gt; coin &lt;= 10</td>
</tr>
<tr>
<td>ROR</td>
<td>ROR_5</td>
<td>if (coin == 10 &amp;&amp; coin != 25 &amp;&amp; coin != 100) {</td>
<td>(line 24) void_coin(int): coin != 10 =&gt; coin == 10</td>
</tr>
<tr>
<td>ROR</td>
<td>ROR_6</td>
<td>if (true &amp;&amp; coin != 25 &amp;&amp; coin != 100) {</td>
<td>(line 24) void_coin(int): coin != 10 =&gt; true</td>
</tr>
<tr>
<td>ROR</td>
<td>ROR_7</td>
<td>if (false &amp;&amp; coin != 25 &amp;&amp; coin != 100) {</td>
<td>(line 24) void_coin(int): coin != 10 =&gt; false</td>
</tr>
<tr>
<td>ROR</td>
<td>ROR_8</td>
<td>if (coin != 10 &amp;&amp; coin &gt; 25 &amp;&amp; coin != 100) {</td>
<td>(line 24) void_coin(int): coin != 25 =&gt; coin &gt; 25</td>
</tr>
<tr>
<td>ROR</td>
<td>ROR_9</td>
<td>if (coin != 10 &amp;&amp; coin &gt;= 25 &amp;&amp; coin != 100) {</td>
<td>(line 24) void_coin(int): coin != 25 =&gt; coin &gt;= 25</td>
</tr>
<tr>
<td>ROR</td>
<td>ROR_10</td>
<td>if (coin != 10 &amp;&amp; coin &lt; 25 &amp;&amp; coin != 100) {</td>
<td>(line 24) void_coin(int): coin != 25 =&gt; coin &lt; 25</td>
</tr>
</tbody>
</table>

### 3.7.4 Results for AOIU operators

*Table 3.7.4 – Results of AOIU operators*

<table>
<thead>
<tr>
<th>Operator</th>
<th>Mutant Name</th>
<th>Mutant Content</th>
<th>Chromosome</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOIU</td>
<td>AOIU_1</td>
<td>credit = -credit + coin;</td>
<td>(line 30) void_coin(int): credit =&gt; -credit</td>
</tr>
<tr>
<td>AOIU</td>
<td>AOIU_2</td>
<td>return -change;</td>
<td>(line 40) int_getChoc(java.lang.StringBuffer):change =&gt; -change</td>
</tr>
</tbody>
</table>
3.7.5 Results for COR operators

Table 3.7.5 – Results of COR operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Mutant Name</th>
<th>Mutant Content</th>
<th>Chromosome</th>
</tr>
</thead>
<tbody>
<tr>
<td>COR</td>
<td>COR_1</td>
<td>if (coin != 10</td>
<td></td>
</tr>
<tr>
<td>COR</td>
<td>COR_2</td>
<td>if (coin != 10 ^ coin != 25 &amp;&amp; coin != 100) {}</td>
<td>(line 24) void _coin(int): coin != 10 &amp;&amp; coin != 100 =&gt; coin != 10 ^ coin != 25</td>
</tr>
<tr>
<td>COR</td>
<td>COR_3</td>
<td>if (coin != 10 &amp;&amp; coin != 25 &amp;&amp; coin != 100) {}</td>
<td>(line 24) void _coin(int): coin != 10 &amp;&amp; coin != 100 =&gt; coin != 10 &amp;&amp; coin != 100 ^ coin != 25</td>
</tr>
<tr>
<td>COR</td>
<td>COR_4</td>
<td>if ((coin != 10 &amp;&amp; coin != 25) ^ coin != 100) {}</td>
<td>(line 24) void _coin(int): coin != 10 &amp;&amp; coin != 100 =&gt; (coin != 10 &amp;&amp; coin != 25 &amp;&amp; coin != 100) ^ coin != 25</td>
</tr>
<tr>
<td>COR</td>
<td>COR_5</td>
<td>if (credit &lt; 90 &amp;&amp; stock.size() &lt;= 0) {}</td>
<td>(line 37) int _getChoc(java.lang.StringBuffer): credit &lt; 90 ^ stock.size() &lt;= 0 =&gt; credit &lt; 90</td>
</tr>
<tr>
<td>COR</td>
<td>COR_6</td>
<td>if (credit &lt; 90 ^ stock.size() &lt;= 0) {}</td>
<td>(line 37) int _getChoc(java.lang.StringBuffer): credit &lt; 90 ^ stock.size() &lt;= 0 =&gt; credit &lt; 90 ^</td>
</tr>
</tbody>
</table>

The above results revealed that the mutants are generated through the whole test suite generation process. It is proved that the algorithm was capable of generating test suite, which is representative of all goals and the branch coverage is given the main focus from these results. The mutation score is computed for the application which tells the coverage dynamics. The lesser in mutation score the higher in code coverage. Mutation testing is one of the testing strategies which also known as the structural testing procedure that takes care of the testing process using the structure of the code. This is the something to deal with the source code in order to modify or improve and then test the code. This process improves the quality of SUT. In the original program, faults are introduced. The program which has been modified to introduce faults is known as a mutant program. When test cases are generated and
applied to both original program and mutants. When the original program and mutant produce same results, the mutant is killed. The following are the formal steps used to perform mutation testing.

- A single syntactic change in the original program is called a mutation. Each mutant program should have different with the original program with only one mutant.
- First of all source code is modified in order to introduce faults. It results in the source code with different versions. When the test case fails with the mutant version, that test case is known as an effective test case.
- If the original program and mutant produce same results, the program can understand it and kill the mutant.
- If the results of the original program and mutant are not same, then that mutant is not killed and used further for making effective test cases.

There are many techniques to create mutants in the program while testing. As the process of modifying the original program with single change or mutation can be done in different ways. Some of the techniques are as follows.

- Replacing an operand with another one is one of the approaches.
- Modification of expressions is another way of making mutants.
- Statements can be modified in order to make mutant programs.
- There are many mutation operators such as
  - Modification of data types
  - Variables’ data modification
  - Replacing a statement
  - Replacement of operators or adding new operators
  - Removing else part in the branching construct.
  - Replacement of the comparable array name
  - Replacement of the logical connector
  - Insertion of unary operator
  - Deletion of a statement
  - Replacing a return statement
  - Replacement of go to label
Some of these experiments are considered in order to make mutants from SUT in this chapter. The generated mutants are used to subject for testing, to improve the quality of test cases generated. The mutation score is computed in this work in order to know the effectiveness. The mutation score is computed as the percentage of killed mutants when compared with the total number of mutants. The following equation is used to know the mutant score.

\[
MS = \frac{(KM)}{(M)} \times 100
\]

Where KM stands for Killed Mutants and M represents the total number of mutants.

In this chapter, experiments are made with multiple Java programs that are written in the object oriented approach. If the mutation score is 100% then the score is said to be mutation adequate. Thus, this chapter achieves the dual goal of generating representative test suite, and also generating mutations in order to measure the adequacy of generated test cases. The advantages of using mutation testing are as follows

- Mutation approach is powerful to achieve high coverage
- Comprehensive testing process
- Increases fault detection ratio
- Software quality is improved
- In the long run customer satisfaction gets improved.
- Ambiguities in the source code of the original program can be detected and thus fault detection ratio gets improved.

3.7.6 LOC of SUT

Different programs used for experiments are named from Prog1 to Prog6. They have different programming structures such as sequence, selection or branching and iteration or looping. The number of lines of code or the LOC metric is employed to know the size of the SUT. The size of the SUT is considered which has a significant impact on the test suite generation time and coverage goals. As shown in Figure 3.7.6.1, the LOC is measured for all the programs and this LOC value considered for the experiments.
As Emphasized in Figure 3.7.6.1, it is evident that the applications are presented in terms of the number of lines of code. Prog 3 has the highest number of lines, while the Prog 1 has the less number of lines of code.

LOC is one of the metrics used to know the size of a computer program. The size does not include blank lines, whitespace and comments. It is one of the widely used measures that is easily understood, simple and widely used in software development firms. Often LOC is also used to estimate the price of software to be developed.

3.7.7 Performance of the Approach with Different SUT

All SUTs are subjected to the generation of whole test suite generation. The time taken to generate a representative test suite is observed and recorded while generating the whole test suite and performing mutation testing which was shown in Figure 3.7.7.1.
Figure 3.7.7.1—Performance with different applications

The SUTs considered for test suite generation exhibit different performance in terms of time taken for test suite generation. The general trend observed in the experiments with the chosen six SUTs. The LOC has its impact on the test suite generation time and also it is stated differently, that the trend tells there is a relation between the LOC and the time needed to generate the representative test suite that covers all branches. As shown in Figure 3.7.7.1, it is evident that there is a performance difference in generating test suites based on the size of the SUT.

3.7.8 Mutant Dynamics of SUTs

Mutant is a change introduced in the source code of SUT. In other words, mutation is the process of intentional introduction of bugs into the source code. This will help to evaluate test cases to know whether they can still work and uncover bugs in the source code. In software engineering, the test engineers follow this approach to evaluate their test cases. Often it is considered to be a good approach to validate test cases generated or manually written. For this reason, it is very important to have mutation testing. The author believes, that the whole test suite needs to be evaluated with mutation testing. Figure 3.7.8.1 shows the dynamics of mutants introduced in SUTs.
Mutants are introduced in the chosen SUTs for finding how the test cases are adequate for testing the applications. The mutants are injected into SUTs based on their size. The number of mutants introduced depended on the LOC of the SUTs. It is observed that the SUT with more branches can have more possible mutants that can help test engineers to know the quality of generated test suites. As can be seen in Figure 3.7.8.1, it is evident that the number of mutants differs from each application.

3.7.9 Average Branch Coverage vs. Average Mutation Score

Branch coverage and mutation score appear to have some relation. Average branch coverage is essential to have multiple test goals to be fulfilled. When all possible branches are covered in the testing process, it is possible to uncover more faults. In the same fashion, the mutant score is to deal with the quality of test cases that have been generated to cover all branches.
The branch coverage can influence possible mutations. As said earlier the size of the SUTs can influence on the branches and the branch coverage can have its impact on the possible mutants. When mutants are introduced, it is possible to know the strength of a test case. With mutants, it is possible to improve the quality of test cases. As can be seen in Figure 3.7.9.1, it is evident that the results reveal the relationships between average branch coverage and the average mutation score using scatter chart. The scatter chart shows data points that are scattered in the graph and relate to both branch coverage and mutation score.

### 3.7.10 Average Branch Coverage vs. Average Mutation Score

Average branch coverage and the average mutation score of the proposed methodology is described here. The concept remains same as the branch coverage and mutations have the relationship and they have mutual influence. Especially when more branches are in the SUT, there is a possibility to have more mutants to improve the quality of generated test cases.

![Counts](image)

**Figure 3.7.10.1 – Average branch coverage vs. average mutation score**

The count is considered the data point that is common for branch coverage and mutation score. The horizontal axis of the scatter chart represents branch coverage while the mutation score is represented on the vertical axis. There is a comparable difference in the proposed approach with respect to the results. The data points relate to both branch coverage and mutation score. The dynamics of the average mutation score and average branch coverage are reflected in the graph. It is evident in the graph
that the results reveal the relationships between average branch coverage and the average mutation score using scatter chart.

3.8 Summary

The Researcher deals with the concept of generation of the representative test suite which ensures complete code coverage in this chapter. Traditional approaches targeted one particular coverage goal. It is evident that the optimization of whole test suite generation is far better than the traditional method of targeting one coverage goal from the recent experiments in software testing. Genetic algorithms are successfully applied to generate unit tests for testing object oriented software. GA is one of the search based algorithms widely used to generate test cases. GA is applied for generating representative test suites and mutation testing in this research. The researcher built a tool to demonstrate concept. Mutation testing became easy with the generation of the test suite that covers all goals. The empirical results revealed that the application is capable of generating whole test suite which is representative of all test goals besides keeping it small in size.